Asqium: A JavaScript Plugin Framework for Extensible Client and Server-Side Components

Vasileios Triglianos(✉) and Cesare Pautasso

Faculty of Informatics, University of Lugano (USI), Lugano, Switzerland
{vasileios.triglianos,cesare.pautasso}@usi.ch
http://asq.inf.usi.ch/

Abstract. JavaScript has become a language for programming complex Web applications, whose logic is deployed across both Web browsers and Web servers. Current software packaging mechanisms for JavaScript enable a basic level of modularity and reuse. However, they have not yet reached full maturity in terms of enabling system extensions with features contributed as third-party plugins, while encapsulating them adequately. In this paper we present a novel plugin system for JavaScript applications, which integrate Node.js modules with HTML5 Web Components. It provides abstractions for: real time and loosely coupled communication between front-end and back-end components, persistent state storage, and isomorphic usage of JavaScript. Plugins can use hooks and events to contribute functionality and embed it into the main application flow, while respecting the common asynchronous non-blocking programming paradigm of JavaScript. We demonstrate the expressiveness of the framework as it is used to build ASQ: an open, extensible educational Web platform.

1 Introduction

Engineering extensible Web applications that span across the server and the client tiers is a challenging task, which can be alleviated by introducing a suitable plugin system. Reusable plugins [1] can thus be properly packaged to be deployed to extend both tiers of the Web application, despite limitations of the current Web technology platform. One limitation is the mismatch between the front and back-end programming languages for implementing the business logic. With the advent of Node.js, a JavaScript runtime environment for server-side applications, the full stack of the application logic can be written in one language. However, using one language for both the client- and server-side components of a plugin is not adequate on its own to efficiently develop, deploy, version and publish the plugin components so that they can be executed on their corresponding hosts. Another issue related to the double scope of such a plugin system is the asynchronous, event-based communication between the client- and server-side components of the plugin.

On the front-end, the main objective is to create encapsulated components, also known as widgets, that may feature User Interface and/or some

P. Cimiano et al. (Eds.): ICWE 2015, LNCS 9114, pp. 81–98, 2015.
DOI: 10.1007/978-3-319-19890-3_7

business logic implemented using HTML5, CSS, and respectively JavaScript. Until recently, Web front-end technologies (e.g., AngularJS, jQuery) dealt poorly with CSS styles and Javascript encapsulation: CSS styles and JavaScript global variables often bled, unintentionally affecting markup and code that was not in the original scope of the author.

On the server-side, the existing Node package manager (`npm`) mechanism for modularizing, packaging and distributing Node.js modules only ensures their dependencies are satisfied. A complete plugin system would provide additional Application Programming Interfaces (API) to interact with the core of the application featuring decoupled interfaces, event-based communication, performance isolation and failure containment, ease of deployment, avoidance of code repetition, all in compliance with the hosting system's conventions and data flow.

While some of the problems mentioned in this section have individual solutions, in this paper we present the first approach that, to our knowledge, implements a JavaScript based plugin system for the client and the server that offers a unified and integrated solution to all of these problems.

The work is motivated by the needs of the ASQ platform [2]. ASQ is a research platform offering a Web-based lecture delivery system that aims to provide presenters with awareness of the audience's comprehension of the presented material. The flow of an ASQ presentation involves presenting slides, asking questions, gathering real time feedback and discussing the results in class; we call this the 'present-ask-answer-assess-feedback' cycle. Questions are an integral part of ASQ, and one of the early design decisions was to allow content authors to create custom question types that can fit into the existing flow and extend it with new functionality, hence the need for a powerful plugin system to make the platform versatile and extensible.

The rest of this paper is structured as follows. In Section 2 we present the main requirements for extensible Web applications. Section 3 discusses the architecture and implementation details of the proposed plugin system, while the motivation for this work and a case study follow in section 4. Finally, we present the state of the art in Section 5 and we conclude this paper with an overview of future research directions for this work in Section 6.

2 Design Goals

From our experience with the design of ASQ and after analyzing the architecture of several modern Web applications featuring real-time updates, we have collected the following required characteristics that should be satisfied by a plugin system to enhance the extensibility of the Web application.

Application Domain Compliance. The domain of each application dictates the respective entities and their privileges, the flow of information between them and the security constraints that govern them. Plugins should adhere to these constraints rules. As an example, in the case of ASQ, the presenter is in control of information flow which follows variations of the *present-ask-answer-assess-feedback*

cycle. A plugin that would automatically show assessment results without the presenter's consent would violate the domain rules.

Open Event Model. Since plugins extend the application with new functionality, it is likely that they may introduce new events which may not be part of the existing 'information flow' events of the application. While the latter are expected to be gracefully handled by a plugin, it is also very important to be able to extend the possible events exchanged within an application with custom ones, as long as they do not violate the main application flow.

Persistence Flexibility. Web applications often use more than one storage technology to tailor the way different parts of their data model are managed. For example, some aggregation operation is performed on some data and the extracted result gets propagated to the clients in real-time; similarly we may also store the raw data for deferred processing. These could result in using a simple fast in-memory key-value storage and slower document-based disk storage respectively. Plugins should be able to take advantage of both strategies.

Encapsulation and Theming. While encapsulation seems like an obvious desirable characteristic which is readily available by most modern programming languages and Web frameworks, until recently it was very hard to create encapsulated front-end components due to limitations of the HTML/CSS platform. Even if the component authors are careful enough to use high specificity CSS selectors there is no guarantee that the rest of the third-party style rules present in a page will not target the widget markup. To avoid bleeding JavaScript global variables [3], developers may choose to use closures. But this approach will make it harder to expose functionality of their widget to third-party code. Some of these problems have workarounds that fail to conceal the fact that JavaScript was not designed for large scale applications. For example the CommonJS and AMD[1] standards allow better encapsulation of JavaScript code at the cost of precompilation which negates the role of JavaScript as an interpreted language. Theming could make matters worse, since rules should either be very specific, which leads to hard-to-maintain codebases; or generic which could result in unintentional style rule leaks towards non-target elements.

Isomorphism. Developing isomorphic [4] Web applications has the benefits of using the same code both on the front-end and the back-end, a milder learning curve for new developers, better communication between front-end and back-end teams and smaller technology stack. JavaScript is the de-facto browser language and demonstrates good asynchronous performance server-side which renders it an ideal candidate for isomorphic applications.

Easy Deployment and Publication. As any piece of software, plugins may go through many iterations and releases, undergoing common steps such as: testing the code in isolation, followed by functional testing within the host system. If everything is complete in terms of target features and successful testing, the plugin gets released so that its users can update to the latest version. This process

[1] Asynchronous Module Definition.

can be tiresome and error prone since there are many file transfers involved and a lot points of failure: moving code between the plugin's source code directory, which in most cases is under some kind of revision control, and the target host system; deploying to remote servers; separating the client-side from the server-side components; and ensuring the compatibility of different versions in plugin-to-plugin dependencies. Streamlining these processes can help both plugin developers and consumers.

3 Plugin Architecture

In this section we present the architecture of our plugin system, asqium, and how it fulfills the design goals of the previous section (Table 1). The proposed design tries to strike a balance between expressiveness and compliance. An increasingly popular approach to achieve expressive freedom without sacrificing compliance to domain constraints is the curation (or screening) of plugins by a dedicated curator team or directly by the community that consumes the plugins [5]. This shifts the weight from the architectural design to the publishing process thus, resulting in less restrictive APIs where most of the restrictions' focus is on creating plugins that are safe (for example to prevent them, where possible, from accidentally deleting data or unintentionally slowing down the User Interface) rather than secure.

The server-side architectural units of our design are: the core and its exposed APIs, the proxy objects and the plugins. The core implements APIs to be consumed by plugins and a very minimal set of business logic. The majority of business logic that is fundamental to the operation of the system is implemented as bundled plugins, as opposed to optional extensions that are implemented as third-party plugins. The exposed APIs are hooks, events, database and settings. The hooks end events APIs undertake the duty of communication and message passing. The database API gives access to the models and the persistence stores of the application. The settings API allows plugins to store and access settings information for a wide range of system and plugin related tasks. Proxy objects act as the interface to the core APIs. The back-end components of a plugin are Node.js modules. On the client-side we distinguish the Event middleware, the core application and the plugins. The core application implements the information flow of the application. It is in control of the WebSocket layer and is also responsible for establishing the initial communication with the front-end components of plugins. The front-end components of plugins are based on the Web Components technology. For the rest of the paper, in the context of plugin components, 'modules' refers to back-end components and 'Web Components' to front-end components (Fig. 1).

3.1 Server-Side Plugins

Server plugins are implemented as npm modules. The only mandatory dependency from the plugin system is extending a base class which offers some conveniences for the developers like declarative mapping of hook names to callbacks

Table 1. Architectural Decisions and Design Goals

Design Goal	Architectural Decisions for Plugins
Application Domain Compliance	Curation of plugins by the community. Hooks to support the information flow of the application.
Open Event Model	Front-end and back-end code of plugins can exchange custom event types.
Persistence Flexibility	Implementation-agnostic persistence API.
Encapsulation and Theming	Web components mitigate the problem of CSS and Javascript bleeding. Custom elements like `core-style` can be shared across plugins to support theming.
Isomorphism	Business logic in both client and server implemented in Javascript
Easy Deployment and Publication	Single repository for both the front- and back-end. Front-end component installed with bower. Back-end component installed with npm.

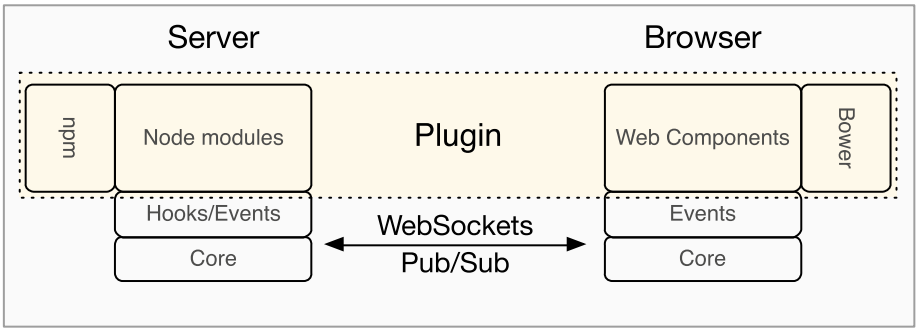

Fig. 1. Plugin structure: Back-end modules and Front-end Web components

and lifecycle methods. There are four lifecycle methods: `install`, `uninstall`, `activate` and `deactivate`, which are called from core when a system user tries to perform one of the corresponding actions. This allows plugins to perform tasks like creating settings, populating the persistent store or performing cleanup.

Proxy Object. Proxy objects are used as a single façade interface between plugin modules and the host system. A proxy object (Fig. 2) exposes all the available APIs that a plugin may use to interface with the core, like hooks, events, settings and database APIs. The implementation of a proxy object is provided by the core. Each time the core instantiates a plugin, it passes the plugin constructor a new instance of a proxy object. Conversely, the core does not directly call methods on plugins: instead it executes hooks or publishes events to which the plugin can subscribe. This ensures that the core remains decoupled from the plugins.

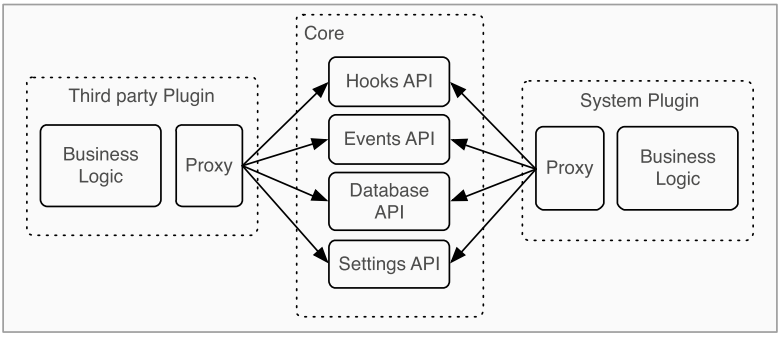

Fig. 2. Proxy object used as the entry point façade for the plugin contributed business logic to access the core APIs

3.2 Front-End Components

Web Components is an umbrella term covering four Web Technologies: Custom elements [6], Shadow DOM [7], the HTML template element and HTML imports [8]. The synthesis of these technologies allows the creation of HTML elements that have encapsulated CSS styles, their own DOM tree (Shadow DOM) and are also JavaScript objects which helps mitigate the global variables bleeding problem which affects many Web applications. Custom elements can be accessed and manipulated with regular DOM methods since they reside and are part of the DOM [9]. Shadow DOM allows us to separate markup that describes content from markup that is purely presentational. Implementation details can be hidden from the user which results to components with succinct markup [10].

We use the Polymer library [11] which builds on top of the Web Components technology. Polymer polyfills [12] missing Web Components Technologies in case they are not available in a browser and adds some functionalities like data-binding and declarative element registration. A typical Web Component defines one or more Custom Elements that encapsulate the User Interface and front-end business logic of the plugin. The plugin subscribes and publishes events to communicate with the rest of the application. Thus, the only dependency between front-end components and the front-end core of the application is the pub/sub implementation which is already present in the form of the EventTarget interface that most DOM Elements implement.

3.3 Communication and Message Passing

Server-Side. We distinguish two modes of server-side task execution and discuss the corresponding communication patterns between participating plugins to accomplish them:

1. **Input Transformation Tasks with Completion Acknowledgement.** These are tasks where each participating plugin applies a transformation to some input. The result is passed to the next participating plugin until there is no plugin

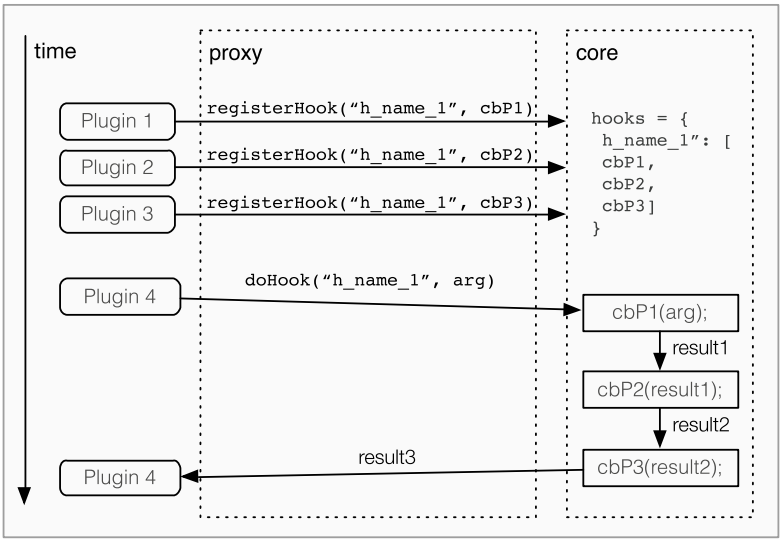

Fig. 3. Hook lifecycle: registration and chained invocation

left in which case the final output is returned to the initiator of the task. Here we only consider the case where order is not important. Participating components can process these tasks in parallel or asynchronously. This is because all participating components are operating on the same data which can lead to unexpected results. Nevertheless, a plugin may execute asynchronous code, which is encouraged for I/O operations, as long as it returns control to the callee when it has finished execution in order to proceed with the next plugin. This can be implemented with callbacks and/or Promises [13]. Examples of such tasks are: knowing when an answer from a student has been processed and persisted to the database in order to update progress information; composing a piece of information that needs to be sent as part of a single HTTP response to the client, like the head of an HTML document.

To target this type of execution, we implemented a hook system. The hook system allows plugins to register for a hook providing the name of the hook and the function (callback) to be invoked when this hook is executed. Hooks identify specific tasks of the Application that require sequential execution and result passing between chained invocations of logic that is contributed by one or more plugins. Hook callbacks have an arity of one, with the only argument being the result of the previous callback execution for the same hook. The initial value for the argument is provided from the initiator of the hook execution similar to a reduce function. Any plugin can initiate a hook execution and any plugin can register. The `doHook` function in Listing 1.1 executes all callbacks for a specific hook. Notice the use of `Promise.reduce` that waits for each task to return either a Promise or a value and then continues with the next callback.

```
function doHook(name){
    if(! this.hookCbs[name]) return Promise.resolve(true);
    var args = Array.prototype.slice.call(arguments, 1);
    //execute callbacks sequentially
    return Promise.reduce(this.hookCbs[name], function(arg, hookFn){
      return Promise.resolve(hookFn(arg));
    }, args);
}
```

Listing 1.1. Executing a hook by triggering contributed callbacks

2. **Decoupled Asynchronous Tasks.** These are tasks that are executed as response to some significant change in the state of the system (event). The initiator of the state change has no knowledge of the other components that are interested in the change. This approach has the benefits of loosely coupled components and asynchronous executions of tasks (which can boost performance [14]). Examples of such state changes that components may want to subscribe to are: 'a new user has come online' or 'a checkbox was ticked'. Event notifications are produced and propagated with message passing according to the pub/sub pattern. A plugin subscribes for an event which may be triggered from the core or other plugins. The dispatcher of the event has no expectations for a completion acknowledgement or some result to be returned. Event-based communication is encouraged in JavaScript development since there's native support in both the server (Node.js - EventEmitter) and the client (DOM - EventTarget).

Client-Side. Our approach is to use pub/sub style communication for plugins and the core using an EventEmitter-like library, and within the plugins a combination of DOM events communication and method invocation. The reason we do not use EventTarget DOM elements as event producers and consumers for non-DOM-related events is to maintain a uniform (isomorphic) interface for event creation and handling across both modules and Web Components. More in detail, the host application awaits for all plugins to be instantiated by listening for a `polymer-ready` event (Listing 1.2). Then it dispatches through the `document` an `app-ready` event for which plugins should have a listener for. As a payload to the event message is the EventEmitter instance that is going to be used as the event bus of the whole application. Plugins can subsequently publish/subscribe to events on the EventEmitter instance (Listing 1.3). The `app-ready` event is the only document dispatched event between a plugin and the application. The rest of the communication is carried out through the EventEmitter.

```
var eventBus = new EventEmitter2();
document.addEventListener("polymer-ready", function(){
  var event = new CustomEvent('app-ready', { 'detail': {appEventBus :
      eventBus} });
  document.dispatchEvent(event);
});
```

Listing 1.2. Seeding the EventEmitter instance to front-end plugins

```
1 document.addEventListener('app-ready', function(evt){
2     evt.detail.appEventBus.on('asq:question_type',
          this.onQuestionType.bind(this));
3 }.bind(this));
```

Listing 1.3. A front-end plugin receiving the EventEmitter instance and using it to subscribe for events

Server-Client Plugin Communication. Real-time Web applications establish low-latency, low-overhead bidirectional communication streams between client and server through WebSockets. Plugins from both sides register for events on an intermediate layer that receives events from the WebSockets layer. The intermediate layer is tasked with filtering events and only re-emitting those that can be consumed by plugins. An EventEmitter-like instance is used to publish or subscribe to events. To send custom messages to the server, Web components use dedicated events, specifying their unique plugin name in the 'type' field of the event. This ensures that modules will process this event since they can subscribe to receive it.

Server generated events targeted to client counterparts of a plugin use the same event structure but an extended identification mechanism. The rationale behind this is that there can be many connected clients that have an instance of the target Web Component and that may be in the scope of a specific event. For example a user may have opened two instances of our application in two separate browser windows; or we may want to target all users that have a specific role in our application, e.g. all the administrators. Our system uses the Socket.IO library to provide WebSocket functionality which allows grouping socket connections in rooms and namespaces [15]. This allows a server plugin to easily target logically grouped clients. Listing 1.9 demonstrates a simple use case of this pattern where a server-side plugin emits an event which targets its client-side counterpart running on clients that belong to the `ctrl` event namespace.

3.4 Persistence

The persistence APIs and behaviour is designed under the assumption that the persistence layer comprises schema-less document or key-value stores like MongoDB and Redis. Enforcing the schema of the data is possible, and recommended, at the business logic layer.

API Interface. Proxy objects provide plugins with an interface to the data stores used by the core. The interface is intended to help to accelerate common development tasks for plugins and mitigate common pitfalls. In our experience, a plugin usually needs two types of persistent data. The first concerns general settings for controlling the plugins' behaviour, for example: activated vs deactivated states, options of the control panel of the plugin, or general configuration options. RDBMS-based applications offer one or more tables dedicated to this cause. RDBMS require data stored in tables to conform to strict schemas.

A common strategy to enable storage of arbitrary data in a single table, is to serialize all data types, including hashmaps and arrays, into strings. Such a concession is not required in document based databases, therefore all types of setting data from all plugins and the core can be stored in one collection (the equivalent of a table)[2].

The second type is data for the plugin business logic. In this case plugin data may either share the same structure as the core generated data or introduce new structures. Again, small mismatches in similar data can be mitigated, as we will demonstrate in the evaluation section, by the schema-less nature of document stores; allowing the plugins to share the same collections as the core. There are cases however, where plugins need to store data that are unrelated with the existing data. In such cases plugins can create their own collections.

Schema and Data Migration. Plugins for traditional Relational Database Management Systems (RDBMS) usually have some logic that migrates the schema and stored data of related database tables from one version of the plugin to another. A lot of modern document based or key-value stores are schema-less and as such there is no need for schema migration. However, for simplicity, data should be kept consistent in structure, so a data migration plan remains necessary. Migration operations can be performed during the `install` lifecycle method of a plugin.

3.5 Plugin Isolation

At runtime, a plugin's code should run in isolation from the code of other plugins and in case of failure, if possible, it should not cause the rest of the application to fail as well. To minimize dependencies and thus the possibility for failures the only way a plugin can interface with the core is through the proxy object instance. The event and hooks systems are agnostic of the presence of plugins. All lifecycle methods of plugin modules that are called from the core are contained in try-catch blocks. This holds true for hooks as well: when a hook is executed the code that initiates the hook execution should catch and handle any errors that may occur from invoking a registered callback contributed by a potentially faulty plugin.

3.6 Deployment and Release Engineering

Npm is the package manager for Node.js modules. It is configured via a `package.json` file. Bower is a package manager for front-end assets like JavaScript libraries and CSS frameworks. It is configured by a `bower.json` file. Our plugin framework uses both to specify the modules and Web components contributed by the plugin respectively (see Fig. 1). The host system can specify the plugin as a dependency using the respective package managers. This design has a number of benefits:

[2] In some cases performance may be affected if some conditions are not met. For example collections with documents that vary greatly in size may induce write penalties.

1. reuses established package managers.
2. allows for a single code repository for the entire plugin.
3. enables to ignore files that are not required by a specific counterpart of the plugin. Npm uses either the `files` field in the `package.json` file for an inclusive list of files or a `.npmignore` file in the project's root directory that lists files to be excluded. Bower supports an `ignore` field in the `bower.json` file to specify which files to exclude.
4. makes it convenient to use the latest version of the plugin code while testing. This can be achieved either by using the tip of a specific code branch or by symlinking the package (be it an npm or a bower package) using built-in package manager commands.

The npm module part of the plugin can be deployed by the command `npm install <package-name>` which will install it in the `node_modules` directory of the server-side code. Given correct implementation of the server-side plugin lifecycle methods, plugins can be installed, uninstalled, activated and deactivated while the application is executing, allowing us to hot-swap implementations while testing. The Web Component part of the plugin can be installed by specifying it as a dependency in the `bower.json` file or the front-end components of the host system. Additional building steps can be implemented to allow code transformation, minification and other release engineering tasks.

4 Evaluation

In this section we present a concrete use case to show that the plugin system delivers the extensibility we need as part of the ASQ [2] project.

4.1 ASQ Application Flow

ASQ is principally aimed at computer science lectures but can be used in any context where presenters require real-time fine grained audience feedback. Presenters add questions to the slides of their presentation, which are implemented in HTML5 and JavaScript, and broadcast the presentation. The audience members connect, follow the slides and answer questions. Continuous feedback is provided to the presenter who may choose to share parts or all of it with the audience. Examples of feedback include incoming answer events, automatic assessment results and more. There might be additional steps involved in the cycle, e.g., peer or self assessment of answers.

Overall, ASQ is designed to take advantage of the plugin system described in this paper to address the following extensibility requirements:

– Content authors should be able to create custom question types or extend existing ones. Types can range from simple multiple choice questions to advanced code editing questions with automatic unit-test assessment.

– Content authors should be able to create custom feedback logic and visualizations to target different presentational needs and accommodate for heterogeneous data coming from different question types. To better illustrate this, let us

assume two different question types: multiple choice questions, where the answer is the combination of the checked options; and a highlight question, where the answer is the highlighted parts of a given text. Whereas it makes sense to render audience responses as a barchart or a pie chart in the former case it may not provide a meaningful visual representation for a heatmap question. A better choice may be to create a heatmap which maps each character position of the text with a color intensity that is proportional to the number of times the specific character position was highlighted by the audience.

– Presenters should be able to enable lecture flow and interaction patterns that match the usage context, their teaching style, the applied pedagogy and the nature of the question types. Examples include: the ability to have synchronized slides between presenter and audience within the classroom and free navigation for viewers during studying (presentation context); the ability to display assessment results in real time versus a specific point in time, or individual versus aggregated results (teaching style); the ability to allow students to work individually or in groups (applied pedagogy); the ability to have automatic, self or peer assessment strategies (applied pedagogy and question type complexity).

– Easily swap data-mining and analytics plugins to compare different techniques and algorithms to increase teacher awareness.

– Easy integration of complex external services and data sources without polluting the system architecture.

4.2 ASQ Architectural Overview

ASQ is a client/server application that uses both HTTP and WebSocket protocols for communication. An overview of the architecture is depicted in figure 4. In the back-end, an HTTP server coupled with the business logic of the application serves static assets, bootstrap data (such as WebSocket connection configuration) and initial HTML which is first rendered server-side using a dynamic template framework. Subsequent client-server communication is WebSocket based for reduced latency and higher throughput. Different roles connect to different 'namespaces', which are pools of connection ids implemented by a software layer on top of WebSockets, albeit in the same host and port.

The business layer persists model data like users, presentations and questions in MongoDB, a document based storage. The application also utilizes a Redis server tasked with two functions: storing simple key-value data like session identifiers and providing a pub/sub implementation that helps scale the WebSocket component of the application.

In the front-end, after an initial page load through HTTP, the application uses mostly WebSocket for communication. The core of the business logic is role-based, so presenters get a different main script than the viewers; and also view-based. For example presenters currently have two discrete views: the 'beamer' view which displays the presenter's version of the presentation and the 'control panel' which displays feedback information like number of connected users, incoming answers, statistics, next and previous slides and more.

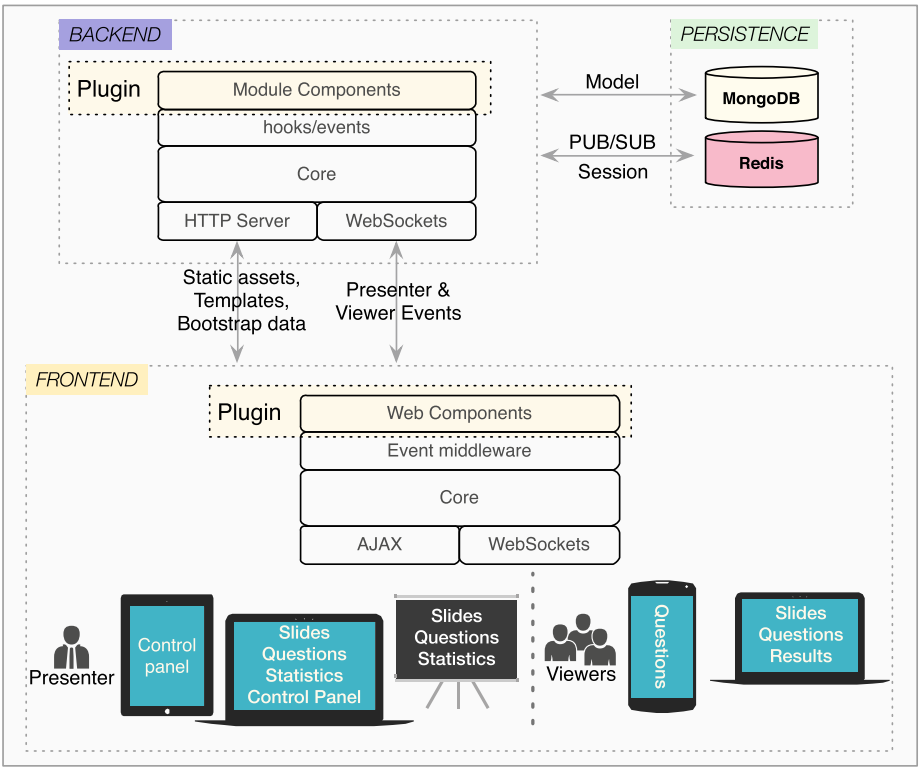

Fig. 4. ASQ architecture with the plugin system

4.3 Crafting a Plugin for ASQ

In this section we demonstrate how we can use the presented plugin system to craft a question type plugin for a highlight question. We will name the plugin 'asq-highlight'.

Creating the User Interface. A custom question type has different appearance and functionality based on the role and view of the user. In our case, an audience viewer faces a text editor with highlight capabilities showing the text to be highlighted (left of Fig. 5). On the beamer view (right of Fig. 5) the presenter can toggle between two modes: a heatmap over all audience answers or the correct solution. On the control panel, the presenter can display the highlight solution for any of the audience members or compare two or more answers with a heatmap.

To implement the User Interface we create a Custom Element, `<asq-highlight>`, that will be the façade for two role-based elements `<asq-highlight-viewer>` and `<asq-highlight-presenter>`; Listing 1.4 shows this in effect. The template of `<asq-highlight>` uses a conditional template (a Polymer feature) to render the appropriate element based on the role attribute

Fig. 5. Viewer (left) and Presenter (right) views of a highlight question

of `<asq-highlight>`. Then it forwards related attributes and the content (also known as distributed nodes) to the role-based elements.

Content authors can embed an `<asq-highlight>` element in their presentation using code similar to Listing 1.5. Notice the definition of two more elements, `<asq-stem>` and `<asq-hl-color-task>`. We can define more than one elements that can be distributed with a Web Component to create structural elements that encapsulate presentation resulting in cleaner and shorter markup.

```
1  <template if="{{role == roles.VIEWER}}">
2    <asq-highlight-viewer mode="{{mode}}" theme="{{theme}}"
         fontSize="{{fontSize}}">
3      <content></content>
4    </asq-highlight-viewer>
5  </template>
6  <template if="{{role == roles.PRESENTER}}">
7    <asq-highlight-presenter mode="{{mode}}" theme="{{theme}}"
         fontSize="{{fontSize}}">
8      <content></content>
9    </asq-highlight-presenter>
10 </template>
```

Listing 1.4. Template of the `<asq-highglight>` element.

```
1  <asq-highlight theme="textmate" mode="java" fontSize="1em">
2     <asq-stem><h3>Highlight with the appropriate color the
          following:</h3></asq-stem>
3     <asq-hl-color-task color="d9534f">Visibility
          Modifiers</asq-hl-color-task>
4     <asq-hl-color-task color="428bca">Variable
          Declarations</asq-hl-color-task>
5     <asq-hl-color-task color="f0ad4e">Other keywords</asq-hl-color-task>
6     <code>public class C {
7    public void m() {
8      int i = i + 5 + ((int)5.0) + ((int)5f);
9    }
10 }</code></asq-highlight>
```

Listing 1.5. Markup to create the showcased highlight question

Implementing Server-Side Business Logic. The server-side module of asq-highlight will respond to two hooks: `parse_html` which is triggered when a user uploads an HTML presentation file; and `answer_submission`, triggered when an audience member submits an answer to a question.

For `parse_html` we are interested in extracting the representation of all `<asq-highlight>` elements present in an HTML string and persisting them in the 'questions' collection of our models persistence store (MongoDB). Listing 1.6 shows the implementation of the callback for `parse_html`. `this.asq` refers to the proxy instance used to persist the extracted question metadata.

```
function parseHtml(html){
  //cheerio is a Node.js library for HTML manipulation
  var   = cheerio.load(html, {decodeEntities: false});
  var hlQuestions = [];
  // extract question metadata from custom elements
   (this.tagName).each(function(idx, el){
    hlQuestions.push(this.processEl( , el));
  }.bind(this));
  //store metadata via persistance API
  return this.asq.db.model("Question").create(hlQuestions)
  .then(function(){
    return Promise.resolve( .root().html());
  });
}
```

Listing 1.6. `parse-html` hook callback for asq-highlight

When a viewer submits an answer to a set of questions, a core plugin fires the `exercise_submission` hook and for each individual question an `answer_submission` hook (Listing 1.7). Asq-highlight provides a callback for `answer_submission` in order to persist the Answer to the database as in Listing 1.8.

Notice the invocation of `this.calculateProgress` which calculates how many of the audience members have answered the question identified by `questionUid` and brodcasts the result to all connected clients. This method executes asynchronously and is a good example of event notification. The abbreviated version in Listing 1.9 shows how the event published from the server can target multiple clients.

```
function handleSubmitEvent(submission){
    //execute 'exercise_submission' hook
    yield hooks.doHook("exercise_submission", submission)
    //execute 'answer_submission' hook for each answer
    yield Promise.map(submission.answers, function(answer){
      return hooks.doHook("answer_submission", answer);
    });
}
```

Listing 1.7. triggering submission hooks in plugins, server-side

```
answerSubmission: coroutine(function *answerSubmissionGen (answer){
  var questionUid = answer.questionUid
  this.validateAnswer(answer);
  yield this.asq.db.model("Answer").create({
    question  : questionUid,
    answeree  : answer.answeree,
    session   : answer.session,
    submitDate : Date.now(),
    submission : answer.submission,
  });
  this.calculateProgress(answer.session, ObjectId(questionUid));
  //this will be the argument to the next hook
  return answer;
}
```

Listing 1.8. submission hook execution

```
var event = {
  questionType: 'asq-highlight',
  type: 'progress',
  questionUid: question_id.toString(),
  heatmapData: JSON.stringify(heatmapData),
}
this.asq.sendSocketEventToNamespaces('asq:question_type', event,
    session_id.toString(), 'ctrl')
```

Listing 1.9. sending an event to all presenter clients

5 Related Work

Architect [16], Intravenous [17], Seneca [18] and Wire [19] are all Node.js architectural frameworks that handle well dependency injection and offer ways to declare modules and their dependencies. They are positioned in the application composition layer on top of the npm modules layer. Front-end components are not in the scope of these frameworks.

This work is heavily influenced by the plugin systems of popular Web-based content management systems. More specifically the hooks construct can be found in similar contexts in Wordpress [20], Ghost [21] and Moodle [22] which are two blogging platforms and an open-source learning platform respectively. Moodle has also the notion of event-driven communication between back-end plugins and the core [23]. In these systems client-server communication between plugin components is performed mainly through HTTP/AJAX. By default, the back-end components cannot push any data to the front-end without the latter having issued an AJAX request first to pull the data [24].

Hoodie [25] is a Node.js framework with CouchDB store technology, whose main goal is to abstract away the back-end to facilitate the job of front-end

developers. To accomplish this, the front-end application communicates with the back-end only through the Hoodie Javascript API. Using CouchDB's changes feed Hoodie is always aware of things that happen to the user's data and makes them available via events which allows keeping multiple devices syncronized. It support plugins which have a: *a*) frontend component; *b*) backend component; and *c*) an admin view. Frontend components communicate with back-end components through tasks similar to the client-server event mechanism described in Section 3.3. A task is a special object that can be saved into the database from the Hoodie front-end. Front-end plugin components deal only with the Hoodie API and do not have visual entities. Any related markup or CSS styles live in the static assets of the main application outside of the plugin directory. Hoodie thus lacks a way to encapsulate markup and styles for front-end plugins. Hoodie also does not have the concept of hooks.

6 Conclusion

In this paper we present the design of asqium: a plugin system for JavaScript/HTML5 Web applications that need to be extended with components running both as back-end modules and as front-end Web components. In addition to achieving the extensibility of the resulting Web application, the plugin system takes care also of basic infrastructural chores, such as event-based communication, persistent storage, and composition of synchronous and asynchronous functions contributed by multiple plugins. The plugin system has been implemented as the foundation of the ASQ educational Web platform, which has provided the motivation for the work and has been used as a case study to evaluate the plugin API expressiveness. We are looking forward to involve the Web Engineering community in further developments. The code for the plugin system implementation is available at https://github.com/ASQ-USI/ASQ/tree/master/lib/plugin. The back-end plugin base is an npm package which can be found at https://github.com/ASQ-USI/asq-plugin.

As a future research direction, we want to explore ways to enable seamless plugin data and state synchronization between different devices and the ability to cache plugin data that are produced when the client is offline and synchronize them upon re-establishing internet connectivity. This will provide a solid foundation on which liquid Web applications [26] can be engineered.

We are also working on ways for plugin authors to prioritize the callback execution associated with hooks and in general specify temporal dependencies between events of different plugins. Finally, we also want to shift our focus towards security and access control. We aim to introduce execution contexts for plugins, that correspond to user-granted privileges, by introducing Role-based Access Control (RBAC) at the plugin level.

Acknowledgments. The work is partially supported by the Swiss Commission for Technology and Innovation with the Spottedmap project (Grant Nr. 16328.1).

References

1. Mayer, J., Melzer, I., Schweiggert, F.: Lightweight plug-in-based application development. In: Aksit, M., Mezini, M., Unland, R. (eds.) NODe 2002. LNCS, vol. 2591, pp. 87–102. Springer, Heidelberg (2003)
2. Triglianos, V., Pautasso, C.: Interactive scalable lectures with ASQ. In: Casteleyn, S., Rossi, G., Winckler, M. (eds.) ICWE 2014. LNCS, vol. 8541, pp. 515–518. Springer, Heidelberg (2014)
3. kangax: Detecting global variable leaks (2009). http://perfectionkills.com/detecting-global-variable-leaks//
4. Bédard, J.: Isomorphic javascript (2015). http://isomorphic.net/
5. Onishi, A.: Plugins: When the time is right. In: Pro WordPress Theme Development, pp. 273–295. Apress (2013)
6. Dimitri, G.: Custom elements. W3c working draft, W3C (December 2014). http://www.w3.org/TR/2014/WD-custom-elements-20141216/
7. Dimitri, G., Ito, H.: Shadow dom. W3c working draft, W3C (June 2014). http://www.w3.org/TR/2014/WD-shadow-dom-20140617/
8. Dimitri, G., Hajime, M.: Html imports. W3c working draft, W3C (March 2014). http://www.w3.org/TR/2014/WD-html-imports-20140311/
9. Penades, S.: An Introduction to Web Components. In: Web Components London, webcomponents.org (January 2015)
10. Walton, P.: Web components and the future of CS. In: Proc. of SFHTML5 (November 2014). http://webcomponents.org/presentations/web-components-and-the-future-of-css/
11. Polymer, P.: Polymer Homepage (2015). https://www.polymer-project.org/
12. Sharp, R.: Detecting global variable leaks (October 2010). https://remysharp.com/2010/10/08/what-is-a-polyfill
13. ECMA: Draft specification for es.next (ecma-262 edition 6). Ecmascript working draft, ECMA (February 2015)
14. Bonetta, D., Binder, W., Pautasso, C.: TigerQuoll: parallel event-based JavaScript. In: Proc. of PPoPP, pp. 251–260 (2013)
15. Rauch, G.: Rooms and Namespaces (2014). http://socket.io/docs/rooms-and-namespaces/ (accessed: February 25, 2015)
16. c9: architect (2015). https://github.com/c9/architect
17. Jacobs, R.: intravenous (2015). https://github.com/RoyJacobs/intravenous
18. Rodger, R.: seneca (2015). https://github.com/rjrodger/seneca
19. cujoJS: wire (2015). https://github.com/cujojs/wire
20. Mullenweg, M., Boren, R., Jaquith, M., Ozz, A., Westwood, P.: Wordpress (2011). https://wordpress.org/
21. Wolfe, H., O'Nolan, J., Davis, P., Williams, J.: Ghost (2015). https://ghost.org/
22. Dougiamas, M.: Moodle: A virtual learning environment for the rest of us. TESL-EJ **8**(2), 1–8 (2004)
23. Moodle developer documentation, M.: Event 2 (2015). https://docs.moodle.org/dev/Event_2 (accessed: February 25, 2015)
24. Mesbah, A., Van Deursen, A.: A component-and push-based architectural style for ajax applications. Journal of Systems and Software **81**(12), 2194–2209 (2008)
25. Hoodie, H.: Hoodie Homepage (2015). http://hood.ie/ (accessed: February 25, 2015)
26. Mikkonnen, T., Systa, K., Pautasso, C.: Towards liquid web applications. In: Proc. of the 15th International Conference on Web Engineering (ICWE). Springer, Rotterdam (2015)